DOCUMENT RESUME

ED 118 130 IR 003 035

AUTHOR	Stonebraker, Michael; And Others
TITLE	The Design and Implementation of INGRES.
INSTITUTION	California Univ., Berkeley. Electronics Research Lab.
SPONS AGENCY	Army Research Office, Durham, N.C.
REPORT NO	UCB-ERL-M-577
PUB DATE	27 Jan 76
NOTE	74p.; Not available in hard copy due to marginal quality of original document; Best copy available
EDRS PRICE	MF-$0.83 Plus Postage. HC Not Available from EDRS.
DESCRIPTORS	Computer Graphics; *Computer Programs; Computers; Data Bases; Electronic Data Processing; *Man Machine Systems; Programing; Programing Languages
IDENTIFIERS	*Data Base Management Systems; INGRES

ABSTRACT

The currently operational version of the INGRES data base management system gives a relational view of data, supports two high level, non-procedural data sublanguages, and runs as a collection of user processes on top of a UNIX operating system. The authors stress the design decisions and tradeoffs in relation to (1) structuring the system into processes, (2) embedding one command language in a general purpose programing language, (3) the algorithms implemented to process in interactions, (4) the access methods implemented, (5) the concurrency and recovery control provided, (6) support for views, protection and integrity constraints, and (7) the data structures used for system catalogs and role of the data base administrator. (Author/CH)

****
* Documents acquired by ERIC include many informal unpublished *
* materials not available from other sources. ERIC makes every effort *
* to obtain the best copy available. Nevertheless, items of marginal *
* reproducibility are often encountered and this affects the quality *
* of the microfiche and hardcopy reproductions ERIC makes available *
* via the ERIC Document Reproduction Service (EDRS). EDRS is not *
* responsible for the quality of the original document. Reproductions *
* supplied by EDRS are the best that can be made from the original. *
****

ED118130

THE DESIGN AND IMPLEMENTATION OF INGRES

by

Michael Stonebraker, Eugene Wong,
Peter Kreps and Gerald Held

U S DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
NATIONAL INSTITUTE OF
EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

Memorandum No. ERL-M577

27 January 1976

BEST COPY AVAILABLE

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

IR 003 035

THE DESIGN AND IMPLEMENTATION OF INGRES

Michael Stonebraker, Eugene Wong and Peter Kreps

Department of Electrical Engineering and Computer Sciences
and the Electronics Research Laboratory
University of California, Berkeley, California 94720

Gerald Held

Tandem Computers, Inc.
Cupertino, California

ABSTRACT

This paper describes the CURRENTLY OPERATIONAL version of the INGRES data base management system. This multi-user system gives a relational view of data, supports two high level non-procedural data sublanguages and runs as a collection of user processes on top of the UNIX operating system for Digital Equipment Corporation PDP 11/40, 11/45 and 11/70 computers. Stressed here are the design decicions and tradeoffs related to 1) structuring the system into processes, 2) embedding one command language in a general purpose programming language, 3) the algorithms implemented to process interactions, 4) the access methods implemented, 5) the concurrency and recovery control provided, 6) support for views, protection and integrity constraints and 7) the data structures used for system catalogs and role of the data base administrator.

Research sponsored by Army Research Office Grant DAHC04-74-G0087, the Naval Electronic Systems Command Contract N00039-76-C-0022, the Joint Services Electronics Program Contract F44620-71-C-0087, the National Science Foundation Grants DCR75-03839 and ENG74-06651-A01, and a Grant from the Sloan Foundation.

1 INTRODUCTION

INGRES (Interactive Graphics and Retrieval System) is a relational data base system which is implemented on top of the UNIX operating system developed at Bell Telephone Laboratories [RITC74] for Digital Equipment Corporation PDP 11/40, 11/45 and 11/70 computer systems. The implementation of INGRES is primarily programmed in "C", a high level language in which UNIX itself is written. Parsing is done with the assistance of YACC, a compiler-compiler available on UNIX [JOHN74].

The advantages of a relational model for data base management systems have been extensively discussed in the literature, [CODD70,CODD74,DATE74] and hardly require further elaboration. In choosing the relational model, we were particularly motivated by (a) the high degree of data independence that such a model affords, and (b) the possibility of providing a high level and entirely procedure-free facility for data definition, retrieval, update, access control, support of views, and integrity verification.

In this paper we will describe the design decisions made in INGRES. In particular, we will stress the design and implementation of:

a) the embedding of all INGRES commands in the general purpose programming language "C"

b) the access methods implemented

c) the catalog structure and the role of the data base administrator

d) support for views, protection and integrity constraints

e) the decomposition procedure implemented

f) implementation of updates and consistency of secondary indices

g) recovery and concurrency control

Except where noted to the contrary, this paper describes the INGRES system operational in January, 1976.

To this end we first briefly describe in Section 1.2 the primary query language supported, QUEL, and the utility commands accepted by the current system. The second user interface, CUPID, is a graphics oriented, casual user language which is also operational and described in [MCDO75a, MCDO75b]. It will not be discussed further in this paper. Then in Section 1.3 we describe the relevant factors in the UNIX environment which have affected our design decisions.

In Section 2 we discuss the structure of the four processes (see Section 1.3 for a discussion of this UNIX notion) into which INGRES is divided and the reasoning behind the choice implemented. The EQUEL (Embedded QUEL) precompiler, which allows the substitution of a user-supplied C program for the "front end" process is also discussed. This program has the effect of embedding all of INGRES in the general purpose programming language "C". Then in Section 3 we indicate the data structures which are implemented in INGRES, the catalog (system) relations which exist and the role of the data base administrator with respect to all relations in a data base. The implemented access methods, their

calling conventions, and the actual layout of data pages in secondary storage where appropriate, are also presented.

Sections 4, 5 and 6 discuss respectively the various functions of each of the three "core" processes in the system. Also discussed are the design and implementation strategy of each process. Lastly, Section 7 draws conclusions, suggests future extensions and indicates the nature of the current applications run on INGRES.

1.2 QUEL AND THE OTHER INGRES UTILITY COMMANDS

QUEL (QUEry Language) has points in common with Data Language/ALPHA [CODD71], SQUARE [BOYC73] and SEQUEL [CHAM74] in that it is a complete [CODD72] query language which frees the programmer from concern for how data structures are implemented and what algorithms are operating on stored data. As such it facilitates a considerable degree of data independence [STON74a].

The QUEL examples in this section all concern the following relation.

	NAME	DEPT	SALARY	MANAGER	AGE
	Smith	toy	10000	Jones	25
EMPLOYEE	Jones	toy	15000	Johnson	32
	Adams	candy	12000	Baker	36
	Johnson	toy	14000	Harding	29
	Baker	admin	20000	Harding	47
	Harding	admin	40000	none	58

Indicated here is an EMPLOYEE relation with domains NAME, DEPT, SALARY, MANAGER and AGE. Each employee has a manager (except for

Harding who is presumably the company president), a salary, an age, and is in a department.

A QUEL interaction includes at least one RANGE statement of the form:

RANGE OF variable-list IS relation-name

The symbols declared in the range statement are variables which will be used as arguments for tuples. These are called TUPLE VARIABLES. The purpose of this statement is to specify the relation over which each variable ranges.

Moreover, an interaction includes one or more statements of the form:

Command [Result-name] (Target-list)
[WHERE Qualification]

Here, Command is either RETRIEVE, APPEND, REPLACE, or DELETE. For RETRIEVE and APPEND, Result-name is the name of the relation which qualifying tuples will be retrieved into or appended to. For REPLACE and DELETE, Result-name is the name of a tuple variable which, through the qualification, identifies tuples to be modified or deleted. The Target-list is a list of the form

Result-domain = Function ...

Here, the Result-domain's are domain names in the result relation which are to be assigned the value of the corresponding function.

The following suggest valid QUEL interactions. A complete description of the language is presented in [HELD75a].

Example 1.1 Find the birth year of employee Jones

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO W (BYEAR = 1975 - E.AGE)
WHERE E.NAME = "Jones"
```

Here, E is a tuple variable which ranges over the EMPLOYEE relation and all tuples in that relation are found which satisfy the qualification E.NAME = "Jones". The result of the query is a new relation, W, which has a single domain, BYEAR, that has been calculated for each qualifying tuple. If the result relation is omitted, qualifying tuples are written in display format on the user's terminal or returned to a calling program in a prescribed format as discussed in Section 2. Also, in the Target-list, the "Result-domain =" may be omitted if Function is the right hand side is an existing domain (i.e. NAME = E.NAME may be written as E.NAME -- see example 1.6).

Example 1.2 Insert the tuple (Jackson,candy,13000,Baker,30) into EMPLOYEE.

```
APPEND TO EMPLOYEE(NAME = "Jackson", DEPT = "candy",
        SALARY = 13000, MGR = "Baker", AGE = 30)
```

Here, the result relation EMPLOYEE is modified by adding the indicated tuple to the relation. If not all domains are specified, the remainder default to zero for numeric domains and null for character strings.

Example 1.3 If a second relation DEPT(DEPT, FLOOR#) contains

the floor# of each department that an employee might work in, then one can fire everybody on the first floor as follows:

```
RANGE OF E IS EMPLOYEE
RANGE OF D IS DEPT
DELETE E WHERE   E.DEPT = D.DEPT
             AND  D.FLOOR# = 1
```

Here E specifies that the EMPLOYEE relation is to be modified. All tuples are to be removed which have a value for DEPT which is the same as some department of the first floor.

Example 1.4 Give a 10 percent raise to Jones if he works on the first floor

```
RANGE OF E IS EMPLOYEE
RANGE of D is DEPT
REPLACE E(SALARY BY 1.1 * E.SALARY)
WHERE E.NAME = "Jones" AND
      E.DEPT = D.DEPT AND D.FLOOR# = 1
```

Here, E.SALARY is to be replaced by 1.1*E.SALARY for those tuples in EMPLOYEE where the qualification is true. (Note that the key-words IS and BY may be used interchangeably with "=" in any QUEL statement.)

Also, QUEL contains aggregation operators including COUNT, SUM, MAX, MIN, and AVG. Two examples of the use of aggregation follow.

Example 1.5 Replace the salary of all toy department employees by the average toy department salary.

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY BY AVG(E.SALARY WHERE E.DEPT = "toy") )
        WHERE E.DEPT = "toy"
```

Here, AVG is to be taken of the salary domain for those tuples satisfying the qualification E.DEPT = "toy". Note that AVG(E.SALARY WHERE E.DEPT= "toy") is scalar valued (in this instance, $13,000) and consequently will be called an AGGREGATE. More general aggregations are possible as suggested by the following example.

Example 1.6 Find those departments whose average salary exceeds the company wide average salary, both averages to be taken only for those employees whose salary exceeds $10000.

```
RANGE OF E IS EMPLOYEE
RETRIEVE INTO HIGHPAY (E.DEPT)
WHERE   AVG(E.SALARY BY E.DEPT WHERE E.SALARY > 10000)
        >
        AVG(E.SALARY WHERE E.SALARY > 10000)
```

Here, AVG(E.SALARY BY E.DEPT WHERE E.SALARY>10000) is an AGGREGATE FUNCTION and takes a value for each value of E.DEPT. This value is the aggregate AVG(E.SALARY WHERE E.SALARY>10000 AND E.DEPT = value). (For the toy, candy and admin departments this value is respectively 14,500, 12,000 and 30,000.) The qualification expression for the statement is then true for departments for which this aggregate function exceeds the aggregate AVG(E.SALARY WHERE E.SALARY>10000).

In addition to the above QUEL commands INGRES also supports a variety of utility commands These utility commands can be classified into seven major categories.

a) invocation of INGRES

INGRES data-base-name

This command executed from UNIX "logs in" a user to a given data base. (A data base is simply a named collection of relations with a given data base administrator who has powers not available to ordinary users.) Thereafter, the user may issue all other commands (except those executed directly from UNIX) within the environment of the invoked data base.

b) creation and destruction of data bases

CREATEDB data-base-name

DESTROYDB data-base-name

These two commands are called from UNIX. The invoker of CREATEDB must be authorized to create data bases (in a manner to be described presently) and he automatically becomes the data base administrator. DESTROYDB successfully destroys a data base only if invoked by the data base administrator.

c) creation and destruction of relations

CREATE relname(domain-name IS format, domain-name IS format,...)

DESTROY relname

These commands create and destroy relations within the current data base. The invoker of the CREATE command becomes the "owner" of the relation created. A user may only destroy a relation that he owns. The current formats accepted by INGRES are 1, 2 and 4 byte integers, 4 and 8 byte floating point numbers and fixed

length ASCII character strings between 1 and 255 bytes.

d) bulk copy of data

```
COPY relname(domain-name IS format, domain-name IS format,...)
  direction  "filename"

PRINT relname
```

The command COPY transfers an entire relation to or from a UNIX file whose name is "filename". Direction is either "TO" or "FROM". The format for each domain is a description of how it appears (or is to appear) in the UNIX file. The relation relname must exist and have domain names identical to the ones appearing in the COPY command. However, the formats need not agree and COPY will automatically convert data types. Also, support is provided for dummy and variable length fields in a UNIX file.

PRINT copies a relation onto the user's terminal formatting it as a report. In this sense, it is stylized version of COPY.

e) storage structure modification

```
MODIFY relname TO storage-structure ON (key1, key2,...)
INDEX ON relname IS indexname(key1, key2,...)
```

The MODIFY command changes the storage structure of a relation from one access method to another. The five access methods currently supported are discussed in Section 2. The indicated keys are domains in relname which are concatenated left to right to form a combined key which is used in the organization of tuples in all but one of the access methods. Only the owner of a

relation may modify its storage structure.

INDEX creates a secondary index for a relation . It has domains of key1, key2,...,pointer. The domain, pointer, is the address of a tuple in the indexed relation having the given values for key1, key2,.... An index named AGEINDEX for EMPLOYEE would be the following binary relation

	AGE	POINTER
	25	address of Smith's tuple
	32	address of Jones' tuple
AGEINDEX	36	address of Adams' tuple
	29	address of Johnson's tuple
	47	address of Baker's tuple
	58	address of Harding's tuple

The relation indexname is in turn treated and accessed just like any other relation except that it is automatically updated when the relation it indexes is updated. This is discussed further in Section 6. Naturally, only the owner of a relation may create and destroy secondary indexes for it.

f) consistency and integrity control.

INTEGRITY CONSTRAINT is qualification

INTEGRITY CONSTRAINT LIST relname

INTEGRITY CONSTRAINT OFF relname

INTEGRITY CONSTRAINT OFF (integer, ... ,integer)

RESTORE data-base-name

The first four commands support the insertion, listing, deletion and selective deletion of integrity constraints which are to be

enforced for all interactions with a relation. The mechanism for handling this enforcement is dicussed in Section 4. The last command restores a data base to a consistent state after a system crash. It must be executed from UNIX and its operation is discussed in Section 6. The RESTORE command is only available to the data-base administrator.

g) miscellaneous

HELP [relname|manual-section]
SAVE relname UNTIL expiration-date
RELKILLER data-base-name

HELP provides information about the system or the data base invoked. When called with an optional argument which is a command name, HELP will return the appropriate page from the INGRES reference manual [ZOOK75]. When called with a relation name as an argument, it returns all information about that relation. With no argument at all it returns information about all relations in the current data base.

SAVE is the mechanism by which a user can declare his intention to keep a relation until a specified time. RELKILLER is a UNIX command which can be invoked by a data base administrator to delete all relations whose "expiration-dates" have passed. This should be done when space in a data base is exhausted. (The data base administrator can also remove any relations from his data base using the DESTROY command, regardless of who their owners are.)

Two comments should be noted at this time.

a) The system currently accepts the language specified as QUEL1 in [HELD75a]. Extension is in progress to accept QUEL2.

b) The system currently does not accept views or protection statements. Although the algorithms have been specified [STON74b,STON75], they have not yet been implemented. For this reason, no syntax for these statements is given in this section; however the subject is dicussed further in Section 4.

1.3 THE UNIX ENVIRONMENT

Two points concerning UNIX are worthy of mention in this section.

a) The UNIX file system

UNIX supports a tree structured file system similar to that of MULTICS. Each file is either a directory (containing references to descendant files in the file system) or a data file. Each data file can be viewed as an array 1 byte wide and 2**24 bytes long. (It is expected that this maximum length will be increased by the UNIX implementors.) Addressing in a file is similar to referencing such an array. Physically, each file is divided into 512 byte blocks (pages). In response to a read request, UNIX moves one or more pages from secondary memory to UNIX core buffers then returns to the user the the actual byte string desired. If the same page is referenced again (by the same or another user) while it is still in a core buffer, no disk I/O takes place.

It is important to note that UNIX pages data from the file system into and out of system buffers using a "least recently used" replacement algorithm. In this way the entire file system is managed as a large virtual store.

In part because the INGRES designers believe that a data base system should appear as a user job to UNIX and in part because they believe that the operating system should deal with all space management issues for the mix of jobs being run, INGRES contains NO facilities to do its own memory management.

Each file in UNIX can be granted by its owner any combination of the following protection clauses:

a) owner read
b) owner write
c) non owner read
d) non owner write
e) execute
f) special execute

When INGRES is initially generated, a UNIX user named INGRES is created. All data files managed by the INGRES system are owned by this "super-user" and have their protection status set to "owner read, owner write, no other access". Consequently, only the INGRES super-user can directly tamper with INGRES files. (The protection system is currently being altered to optionally require the consent of the data base administrator before unrestricted access by the super-user is allowed.)

The INGRES object code is stored in files whose protection status is set to "special execute, no other access". When a user

invokes the INGRES system (by executing command a) above), UNIX creates the INGRES processes operating temporarily with a user-id of INGRES. When a user exits from INGRES these processes are destroyed and the user is restored to operating with his own user-id.

Using this mechanism, the only way a user may access an INGRES data base is to execute INGRES object code. This "safety latch" effectively isolates users from tampering directly with INGRES data.

b) The UNIX process structure

A process in UNIX is an address space (64K bytes or less on an 11/40, 128K bytes or less on 11/45's and 11/70's) which is associated with a user-id and is the unit of work scheduled by the UNIX scheduler. Processes may "fork" subprocesses; consequently, a parent process can be the root of a process subtree. Furthermore, a process can request that UNIX execute a file in a descendant process. Such processes may communicate with each other via an inter-process communication facility called "pipes". A pipe may be declared as a one direction communication link which is written into by one process and read by a second one. UNIX maintains synchronization of pipes so no messages are lost. Each process has a "standard input device" and a "standard output device". These are usually the user's terminal but may be redirected by the user to be files, pipes to other processes, or other devices.

Lastly UNIX provides a facility for processes executing re-entrant code to share procedure segments if possible. INGRES takes advantage of this facility so the core space overhead of multiple concurrent users is only that required by data segments.

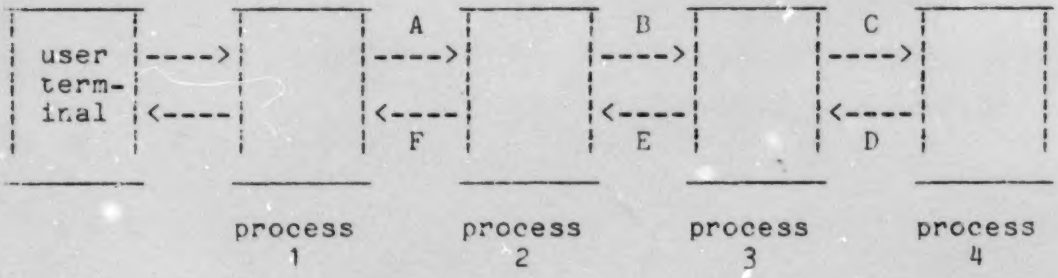
We turn in the next section to the process structure in which INGRES runs.

2 THE INGRES PROCESS STRUCTURE

INGRES can be invoked in two ways: First, it can be directly invoked from UNIX by executing INGRES data-base-name; second it can be invoked by executing a program written using the EQUEL precompiler. We discuss each in turn and then comment briefly on why two mechanisms exist.

2.1 INVOCATION FROM UNIX

Issuing INGRES as a UNIX command causes the process structure shown in Figure 1 to be created.



INGRES Process Structure

Figure 1

Process 1 is an interactive terminal monitor which allows the user to formulate, print, edit and execute collections of INGRES commands. It maintains a workspace with which the user interacts until he is satisfied with his interaction. The contents of this workspace are passed down pipe A as a string of ASCII characters when execution is desired.

As noted above, UNIX allows a user to alter the standard input

and output devices for his processes when executing a command. As a result the invoker of INGRES may direct the terminal monitor to take input from a user file (in which case he runs a "canned" collection of interactions) and direct output to another device (such as the line printer) or a file.

The current terminal monitor accepts the following commands. Anything else is simply appended to the user's workspace.

- # : Erase the previous character. Successive uses of this instruction will erase back to, but not beyond, the beginning of the current line.
- @ : Erase the current line. Successive uses of this instruction are ignored.
- \r : Erase the entire interaction (reset the workspace). The former contents of the workspace are irretrieveably lost.
- \p : Print the current workspace. Its contents are printed on the user's terminal.
- \e : Enter the UNIX text editor and begin accepting editor commands. The editor allows sophisticated editing of the user's workspace. This command is executed by simply "forking" a subprocess and executing the UNIX editor in it.
- \g : Process the current query (go). The contents of the workspace are transmitted to process 2.
- \q : Exit from INGRES.

Process 2 contains a lexical analyzer, a parser, query modification routines for integrity control (and in the future support of views and protection) and concurrency control. When process 2 finishes, it passes a string of tokens to process 3 through pipe B. Process 2 is discussed in Section 4.

Process 3 accepts this token string and contains execution

routines for the commands RETRIEVE, REPLACE, DELETE and APPEND. Any update is turned into a RETRIEVE command to isolate tuples to be changed. Revised copies of modified tuples are spooled into a special file. This file is then processed by a "deferred update processor" in process 4 which is discussed in Section 6.

Basically process 3 performs two functions for RETRIEVE commands. a) A multivariable query is DECOMPOSED into a sequence of interactions involving only a single variable. b) A one variable query is executed by a one variable query processor (OVQP). OVQP in turn performs its function by making calls on the access methods. These two functions are discussed in Section 5; the access method are indicated in Section 3.

In process 4 resides all code to support utility commands (CREATE, DESTROY, INDEX, etc.). Process 3 simply passes to process 4 any commands which process 4 will execute. Process 4 is organized as a collection of overlays which accomplish the various functions. The structure of this process will be discussed in Section 6.

Error messages are passed back through pipes D, E and F to process 1 which returns them to the user. If the command is a RETRIEVE with no result relation specified, process 3 returns qualifying tuples in a stylized format directly to the "standard output device" of process 1. Unless redirected, this is the user's terminal.

We now turn to the operation of INGRES when invoked by code from

the precompiler.

2.2 EQUEL

Although QUEL alone provides the flexibility for most data management requirements, there are many applications which require a customized user interface in place of the QUEL language. For this as well as other reasons, it is often useful to have the flexibility of a general purpose programming language in addition to the data base facilities of QUEL. To this end, a new language, EQUEL (Embedded QUEL), has been implemented which consists of QUEL embedded in the general purpose programming language "C".

In this section we describe the EQUEL language and indicate how it operates in the INGRES environment.

In the design of EQUEL, the following goals were set:

1) The new language must have the full capabilities of both "C" and QUEL.
2) The C program should have the capability for processing each tuple individually which satisfies the qualification in a QUEL RETRIEVE statement. (this is the "piped" return facility described in Data Language/ALPHA [CODD71]).
3) The implementation should make as much use as possible of the existing C and QUEL language processors. (The implementation cost of EQUEL should be small).

With these goals in mind, EQUEL was defined as follows:

1) Any C language statement is a valid EQUEL statement.

2) Any QUEL statement (or INGRES utility command) is a valid EQUEL statement as long as it is prefixed by two number signs ("##").

3) C program variables may be used in QUEL statements in place of relation names, domain names, target list elements, or domain values. The declaration statements of C variables used in this manner must also be prefixed by double number signs.

4) RETRIEVE statements without a result relation have the form

RETRIEVE (Target-list)

[WHERE Qualification] ##{ C-block ##}

which results in the C-Block being executed once for each qualifying tuple.

Two short examples illustrate EQUEL syntax.

Example 2.1 The following section of code implements a small front end to INGRES which performs only one query. It reads in the name of an employee and prints out the employee's salary in a suitable format. It continues to do this as long as there are more names to be read in. The functions READ and PRINT are assumed to have the obvious meaning.

```
main()
{
## char NAME[20];
## int SAL;
while (READ(NAME))
        {
##      RANGE OF X IS EMP
##      RETRIEVE (SAL = X.SALARY)
##      WHERE X.NAME = NAME
```

```
##              {
                PRINT("The salary of ",NAME," is ",SAL);
##              }
        }
}
```

In this example the C-variable NAME is used in the qualification of the QUEL statement and for each qualifying tuple, the C-variable SAL is set to the appropriate value and then the Print statement is executed. (note: in C "{" and "}" are equivalent to BEGIN and END in ALGOL).

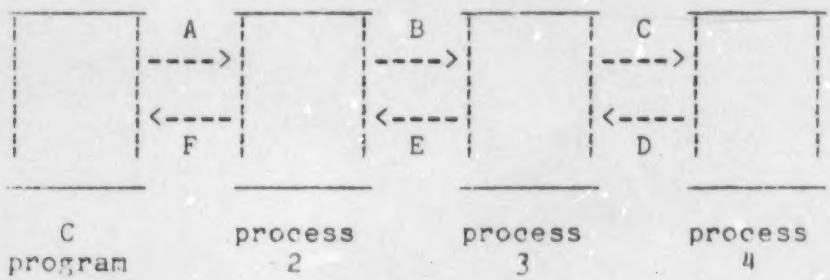
Example 2.2 Read in a relation name and two domain names. Then for each of a collection of values which the second domain is to assume, do some processing on all values which the first domain assumes. (We assume the functions READ and PROCESS exist and have the obvious meanings.) A more elaborate version of this program could serve as a simple report generator.

```
## int VALUE;
## char RELNAME[13], DOMNAME[13], DOMVAL[80];
## char DOMNAME_2[13];
READ(RELNAME);
READ(DOMNAME);
READ(DOMNAME_2);
## RANGE OF X IS RELNAME
while (READ(DOMVAL))
        {
##      RETRIEVE (VALUE = X.DOMNAME)
##          WHERE X.DOMNAME_2 = DOMVAL
                {
                PROCESS(VALUE);
                }
        }
```

Any RANGE declaration (in this case the one for X) is assumed by INGRES to hold until redefined. Hence, only one RANGE statement

is required regardless of the number of times the RETRIEVE statement is executed.

In order to implement EQUEL, a translator (pre-compiler) was written which converts an EQUEL program into a valid C-program with QUEL statements converted to appropriate C-code and calls to INGRES. The resulting C-program is then compiled by the normal C-compiler producing an executable module. Moreover, when an EQUEL program is run, the executable module produced by the C-compiler is used as the front end process in place of the interactive terminal monitor as noted in Figure 2.



The Forked Process Structure

Figure 2

During execution of the front-end program, data base requests (QUEL statements in the EQUEL program) are passed through pipe A and processed by INGRES. If tuples must be returned for tuple at a time processing, then they are returned through a special data pipe set up between process 3 and the C program. A condition code is also returned through pipe F to indicate success or the type of error encountered.

Consequently, the EQUEL translator must perform the following five functions:

1) insert system calls to "spawn" at run time the process structure shown in Figure 2.

2) note C-variable declarations prefaced by ## as legal for inclusion in INGRES commands.

3) process other lines prefaced by ##. These are parsed to isolate C-variables. In adddition, C statements are inserted to write the line down pipe A in ASCII format, modified so that values are substituted for any C-variables. The rationale for not completely parsing a QUEL statement in EQUEL is given in [ALLM76].

4) insert C statements to read pipe F for completion information and call the procedure IIerror. The user may define IIerror himself or have EQUEL include a standard version which prints the error message (for abnormal terminations) and continues.

5) If data is to be returned through the data pipe (by a RETRIEVE with no result relation specified), EQUEL must also:

a) insert C statements to read the data pipe for a tuple formatted as type/value pairs.

b) insert C statements to substitute values into C-

variables declared in the target list. If necessary, values are converted to the types of the declared C-variables.

c) insert C statements to pass control to the C-block following the RETRIEVE.

d) insert C statements following the block to return to step a) if there are more tuples.

2.3 COMMENTS ON THE PROCESS STRUCTURE

The process structure shown in Figures 1 and 2 is the fourth different process structure implemented. The following considerations suggested this final choice:

a) Simple control flow. Previous process structures had a more complex interconnection of processes which made debugging harder.

b) Commands are passed to the right only. Process 3 must issue commands to various overlays in process 4 to execute interactions as discussed in Section 5. Hence, process 3 must be to the left of process 4.

c) The utility commands are expected to be called relatively infrequently compared to the activity in process 2 and 3. Hence, it appears appropriate to overlay little used code in a single process. The alternative is to create additional processes (and pipes) which are quiescent most of the time. This would require added space in UNIX core tables for no particular advantage.

d) The first 3 processes are used frequently. Overlaying code in these processes was tried in a previous version and slowed the system considerably.

e) To run on an 11/40, the 64K address space limitation must be adhered to. Processes 2 and 3 are nearly their maximum size and hence cannot be combined. (For 11/45 and 11/70 versions we may experiment with such a combination.)

f) The C program which replaces the terminal monitor as a front end must run with a user-id different from that of INGRES for protection reasons. (Otherwise it could tamper directly with data managed by INGRES.) Hence, either it must be overlayed into a process or run in its own process. For efficiency and convenience, the latter was chosen.

g) The interactive terminal monitor could have been written (albeit clumsily) in EQUEL. Such a strategy would have avoided the existence of two process structures which differ only by the treatment of the data pipe. This was not done because response time would have degraded and because EQUEL does type conversion to predefined types. This feature would unnecessarily complicate the terminal monitor.

h) The processes are all synchronized (i.e. each waits for an error return from the next process to the right before continuing to accept input from the process to the left. This is done because it simplifies the flow of control. Moreover in many instances the various processes MUST be synchronized. Future

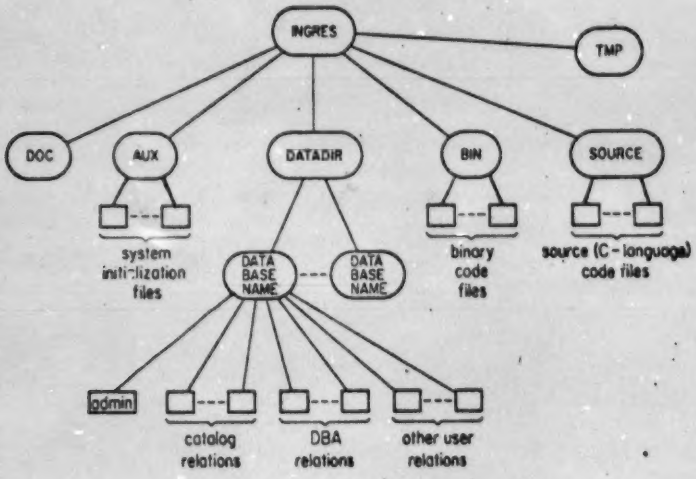
versions of INGRES may attempt to exploit parallelism where possible.

3 DATA STRUCTURES AND ACCESS METHODS

We begin this section with a discussion of the files that INGRES manipulates and their contents. Then we sketch the language used to access all non directory files. Finally, the five possible file formats are indicated.

3.1 THE INGRES FILE STRUCTURE

Figure 3 indicates the subtree of the UNIX file system that INGRES manipulates.



The INGRES Subtree

Figure 3

The root of this subtree is a directory made for the UNIX user "INGRES". It has six descendant directories. The AUX directory contains descendant files containing tables which control the spawning of processes shown in Figures 1 and 2, and an authorization list of users who are allowed to create data bases. Only

the INGRES "super-user" may modify these files (by using the UNIX editor). BIN and SOURCE are directories indicating descendant files of respectively object and source code. TMP contains temporary files containing the workspaces used by the interactive terminal monitor. DOC is the root of a subtree with system documentation and the reference manual. Lastly there is a directory entry in DATADIR for each data base that exists in INGRES. These directories contain the data base files in a given data base as descendants.

These data base files are of four types:

a) an administration file. This contains the user-id of the data base administrator (DBA) and initialization information.

b) System relations. These relations have predefined names and are created for every data base. They are owned by the DBA and constitute the system catalogs. They may be queried by a knowledgeable user issuing RETRIEVE statements, however, they may be updated only by the INGRES utility commands (or directly by the INGRES "super-user" in an emergency). (When protection statements are implemented the DBA will be able to selectively restrict RETRIEVE access to these relations if he wishes.) The form and content of some of these relations will be presently discussed.

c) DBA relations. These are relations owned by the DBA and are shared in that any user may access them. When protection is implemented the DBA can "authorize" other users by inserting

protection predicates (which will be in one of the system relations) and "deauthorize" them by removing such predicates.

d) Other relations. These are relations created by other users (by RETRIEVE into W or CREATE) and are NOT SHARED.

Three comments should be made at this time.

a) The DBA has the following power not available to ordinary users:

1) the ability to create shared relations and to specify access control for them

2) the ability to run RELKILLER

3) the ability to destroy any relations in his data base (except the system catalogs)

This system allows "one level sharing" in that only the DBA has the powers in a) and he cannot delegate any of these powers to others (as in the file systems of most time-sharing systems). This strategy was implemented for three reasons:

1) The need for added generality was not perceived. Moreover, added generality would have created tedious problems (such as making revocation of access privileges non trivial).

2) It seems appropriate to entrust to the DBA the duty (and power) to resolve the policy decision which must be

made when space is exhausted and some relations must be destroyed (or archived). This policy decision becomes much harder (or impossible) if a data base is not in the control of one user.

3) Someone must be entrusted with the policy decision concerning which relations to physically store and which to define as "views". This "data base design" problem is best centralized in a single DBA.

b) Except for the single administration file in each data base every file is treated as a relation. Storing system catalogs as relations has the following advantages:

1) Code is economized by sharing routines for accessing both catalog and data relations.

2) Since several storage structures are supported for accessing data relations quickly and flexibly under various interaction mixes, these same storage choices may be utilized to enhance access to catalog information.

3) The ability to execute QUEL statements to examine (and patch) system relations where necessary has greatly aided system debugging.

c) Each relation is stored in a separate file, i.e., no attempt is made to "cluster" tuples from different relations which may be accessed together on the same (or a nearby) page. This decision is based on the following reasoning.

1) The access methods would be more complicated if clustering were supported.

2) UNIX has a small (512 byte) page size. Hence it is expected that the number of tuples which can be grouped on the same page is small. Moreover, logically adjacent pages in a UNIX file are NOT NECESSARILY physically adjacent. Hence clustering tuples on "nearby" pages has no meaning in UNIX; the next logical page in a file may be further away (in terms of disk arm motion) than a page in a different file. In keeping with the design decision of NOT modifying UNIX, these considerations were incorporated in the design decision not to support clustering.

3) Clustering of tuples only makes sense if associated tuples can be linked together using "sets" [CODA71] or "links" [TSIC75]. Incorporating these access paths into the decomposition scheme would have greatly increased its complexity.

3.2 SYSTEM CATALOGS

We turn now to a discussion of the system catalogs. We discuss two relations in detail and indicate briefly the contents of the others.

The RELATION relation contains one tuple for every relation in the data base (including all the system relations.) The domains of this relation are:

relid	the name of the relation
owner	the UNIX user-id of the relation owner; when appended to relid it produces a unique file name for storing the relation.
spec	indicates one of 5 possible storage schemes or else a special code indicating a virtual relation (or "view").
indexd	flag set if secondary index exists for this relation. (This flag and the following two are present to improve performance by avoiding catalog lookup's when possible during query modification and one variable query processing.)
protect	flag set if this relation has protection predicates.
integ	flag set if there are integrity constraints.
save	scheduled life time of relation.
tuples	number of tuples in relation.
atts	number of domains in relation.
width	width (in bytes) of a tuple.
prim	number of primary file pages for this relation.

The ATTRIBUTE catalog contains information relating to individual domains of relations. Tuples of the ATTRIBUTE catalog contain

the following items for each domain of every relation in the data base:

relid	name of relation in which attribute appears
owner	relation owner
domain-name	domain name
domainno	domain number (position) in relation. In processing interactions INGRES uses this number to reference this domain.
offset	offset in bytes from beginning of tuple to beginning of domain.
type	data type of domain (integer, floating point or character string).
length	length (in bytes) of domain;
keyno	if this domain is part of a key, then "keyno" indicates the ordering of this domain within the key.

These two catalogs together provide information about the structure and content of each relation in the data base. No doubt items will continue to be added or deleted as the system undergoes further development. The first planned extensions are the minimum and maximum values assumed by the domain. These will be used by a more sophisticated decomposition scheme being developed, which is discussed briefly in the next section and in detail in [WONG76]. The representation of the catalogs as relations has allowed this restructuring to occur very easily.

Several other system relations exist which provide auxiliary information about relations. The INDEX catalog contains a tuple for every secondary index in the data base. Since secondary indices are themselves relations they are independently cataloged in the RELATION and ATTRIBUTE relations. However, the INDEX catalog provides the association between a primary relation and the secondary indices for it including which domains of the primary relation are in the index.

The PROTECTION and INTEGRITY catalogs contain respectively the protection and integrity predicates for each relation in the data base. These predicates are stored in a partially processed form as character strings. (This mechanism exists for INTEGRITY and will be implemented in the same way for PROTECTION.) The VIEW catalog will contain, for each virtual relation, a partially processed QUEL-like description which can be used to construct the view from its component physical relations. The use of these last three catalogs will be described in Section 4. The existence of any of this auxiliary information for a given relation is signalled by the appropriate flag(s) in the RELATION catalog.

Yet another set of system relations are those used by the graphics sub-system to catalog and process maps, which (like everything else) are stored as relations in the data base. This topic has been discussed separately in [GO75].

3.3 ACCESS METHODS INTERFACE (AMI).

We will now discuss in more detail the AMI which handles all

actual accessing of data from relations. The AMI language is implemented as a set of functions whose calling conventions are indicated below.

Each access method must do two things to support the following calls. First it must provide SOME linear ordering of the tuples in a relation so that the concept of "next tuple" is well defined. Second it must assign to each tuple a tuple-id (TID) which uniquely identifies a tuple.

The nine implemented calls are as follows:

a) openr(descriptor, mode, relation_name)

Before a relation may be accessed it must be "opened". This function opens the UNIX file for the relation and fills in a "descriptor" with information about the relation from the RELATION and ATTRIBUTE catalogs. The descriptor, which must be declared in the calling program, is used in subsequent calls on AMI routines as an input parameter to indicate what relation is involved. Consequently, the AMI data accessing routines need not themselves check the system catalogs for the description of a relation. "Mode" specifies whether the relation is being opened for update or for retrieval only.

b) get(descriptor, tid, limit_tid, tuple, next_flag)

This function retrieves into 'tuple' a single tuple from the relation indicated by 'descriptor'. 'tid' and 'limit_tid' are

tuple-identifiers. There are two modes of retrieval, "scan" and "direct". In "scan" mode "get" is intended to be called successively to retrieve all tuples within a range of tuple-id's. An initial value of 'tid' sets the low end of the range desired and 'limit_tid' sets the high end. Each time "get" is called with 'next_flag' = TRUE, the tuple following 'tid' is retrieved and its tuple-id placed into 'tid' in readiness for the next call. Reaching 'limit_tid' is indicated by a special return code. The initial setting of 'tid' and 'limit_tid' is done by the "find" function. In "direct" mode ('next_flag' = FALSE) the function retrieves the tuple with tuple-id = 'tid'.

c) find(descriptor, key, tid, match_mode)

"Find" places in 'tid' the tuple-id at the low or high end of the range of tuples which match the key value supplied. The matching condition to be applied depends on 'match-mode'.

If the relation does not have a keyed storage structure or if the key supplied does not correspond to the correct key domains, the 'tid' returned will be as if no key were supplied. The objective of "find" is to restrict the scan of a relation by eliminating from consideration those tuples known from their placement in the relation not to satisfy the matching condition with the key. Calls to "find" occur in pairs, one to set the low end of a scan, the other for the high end, and the two tuple-id's obtained are used in subsequent calls on "get".

Two functions are available for determining the access

characteristics of the storage structure of a primary data relation or secondary index, respectively.

d) paramd(descriptor, access_characteristics_structure)

e) parami(descriptor, access_characteristics_structure)

These functions fill in the 'access_characteristics_structure' with information regarding the type of key which may be constructed to optimize access to the given relation. This includes whether exact key values or ranges of key values can be used, and whether a partially specified key may be used. This will determine the 'match-mode' used in a subsequent call to "find". The ordering of domains in the key is also indicated. These functions relieve optimization routines executed during the processing of an interaction of the need to know directly about specific storage structures.

Other AMI functions provide a facility for updating relations.

f) insert(descriptor, tuple)

The tuple is added to the relation in its "proper" place according to its key value and the storage mode of the relation.

g) replace(descriptor, tid, new_tuple)

h) delete(descriptor, tid)

The tuple indicated by 'tid' is either replaced by new values or deleted from the relation altogether. The tuple-id of the affected tuple will have been obtained by a previous "get".

Finally, when all access to a relation is complete it must be closed:

i) closer(descriptor)

This closes the relation's UNIX file and rewrites the information in the descriptor back into the system catalogs if there has been any change.

3.4 STORAGE STRUCTURES AVAILABLE.

We will now describe the five storage structures currently available in INGRES. Four of the schemes are keyed, i.e., the storage location of a tuple within the file is a function of the value of the tuple's key domains. These schemes allow rapid access to specific portions of a relation when key values are supplied. The remaining non-keyed scheme stores tuples in the file independently of their values and provides a low-overhead storage structure, especially attractive when the entire relation must be accessed anyway.

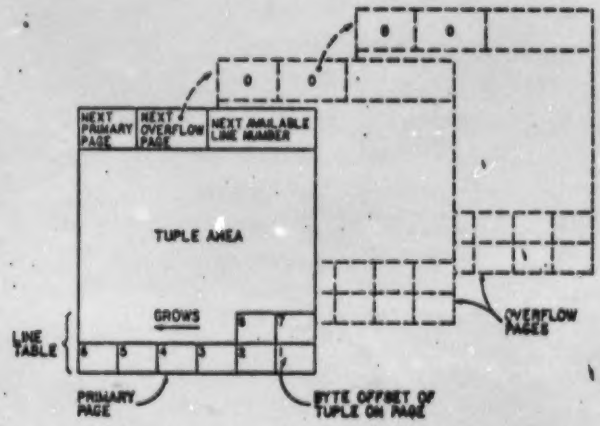
The non-keyed storage structure in INGRES is a randomly ordered sequential file. Fixed-length tuples are simply placed sequentially in the file in the order supplied. New tuples added to the relation are merely appended to the end of the file. The unique tuple-identifier for each tuple is its byte-offset within the file. This mode is intended mainly for

a) very small relations, for which the overhead of other schemes is unwarranted

b) transitional storage of data being moved into or out of the system by COPY;

c) certain temporary relations created as intermediate results during query processing.

In the remaining schemes, the key-value of a tuple determines the page of the file on which the tuple will be placed. The schemes share a common "page-structure" for managing tuples on file pages as shown in Figure 4.



Page Layout for Keyed Storage Structures

Figure 4

A tuple must fit entirely on a single page. Its unique identifier (TID) consists of a page number (the ordering of its page in the UNIX file) plus a "line number" indicating its position on the page. A "line table", which grows upwards from the bottom of the page, contains as an entry for each tuple, a pointer to the beginning of the tuple. In this way a page can be reorganized without affecting TID's.

Initially the file will contain all its tuples on a number of "primary" pages. If the relation grows and these pages fill, "overflow" pages are allocated and chained by pointers to the primary pages with which they are associated. Within a chained group of pages no special ordering of tuples is maintained. Thus in a keyed access which locates a particular primary page, tuples matching the key may be on any page in the chain.

As discussed in [HELD75b] two modes of key-to-address transformation are used -- randomizing and order preserving. In a "hash" file tuples are distributed randomly throughout the primary pages of the file according to a hashing function on a key. This mode is well suited for situations in which access is to be conditioned on a specific key value.

As an order-preserving mode, a scheme similar to IBM's ISAM [IBM66] is used. The relation is sorted to produce the ordering on a particular key. A multi-level directory is created which records the high key on each primary page. The directory, which is static, resides on several pages within the file itself, following the primary pages. A primary page and its overflow pages are not maintained in sort order. This decision is discussed in the section on concurrency. The "ISAM-like" mode is useful in cases where the key value is likely to be specified as falling within a range of values, since a near ordering of the keys is preserved. The index compression scheme discussed in [HELD75b] is currently under implementation.

In the above mentioned keyed modes, fixed length tuples are stored. In addition, both schemes can be used in conjunction with data compression techniques [GOTT75] in cases where increased storage utilization outweighs the added cost of encoding and decoding data during access. These modes are known as "compressed hash" and "compressed ISAM".

The current compression scheme suppresses blanks and portions of a tuple which match the preceding tuple. This compression is applied to each page independently. Other schemes are being experimented with.

3.5 ADDITION OF NEW ACCESS METHODS

One of the goals of the AMI design was to insulate higher level software from the actual functioning of the access methods and thereby to make it easy to add different ones.

In order to add a new access method one need only extend the AMI routines to handle the new case. If the new method uses the same page layout and TID scheme, only find, parami, and paramd need to be extended. Otherwise new procedures to perform these functions must be supplied for use by get, insert, replace and delete.

4 THE STRUCTURE OF PROCESS 2

Process 2 contains code to perform four main functions

a) a lexical analyzer

b) a parser (written in YACC [JOHN74])

c) query modification routines to support protection, views and integrity control

d) concurrency control

These are discussed in turn

4.1 LEXICAL ANALYSIS AND PARSING

The lexical analysis and parsing phases of INGRES have been organized around the YACC translator writing system available in UNIX [JOHN74]. YACC takes as input a description of a grammar consisting of BNF-like parsing rules (productions) and precedence rules, plus action statements associated with each production. It produces a set of tables to be interpreted by a parse table interpreter which is combined with locally-supplied lexical analysis and parsing action routines to produce a complete translator.

The interpreter uses a bottom-up LR(1) parsing approach. The lexical analyzer is called to obtain successive symbols from the input stream as the interpreter attempts to match the input with productions in the grammar. When a production is matched YACC performs a reduction and executes the action statement associated with the production. YACC has a mechanism for recovering from

errors to continue parsing input in its entirety.

While the YACC parse table interpreter checks the syntactic correctness of the input commands, the action statements check for semantic consistency and correctness and prepare the commands for further processing. The system catalogs are used to check that relation and domain names, formats, and so on, are specified appropriately.

For utility commands a command indicator and the parameters for the command are sent directly to process 3 for transmission to process 4. Section 6 discusses these commands and their implementation.

For QUEL commands, the input is translated to a tree-structured internal form which will be used in the remaining analysis and processing. Moreover, the qualification part is converted to conjunctive normal form. The parse tree is now ready to undergo what has been termed "query-modification," to be described in Section 4.2 and 4.3.

4.2 INTEGRITY

Query modification includes adding integrity and protection predicates to the original query and changing references to virtual relations into references to the appropriate physical relations. At the present time only a simple integrity scheme has been implemented.

In [STON75] algorithms of several levels of complexity are

presented for performing integrity control on updates. In the present system only the simplest case, involving single-variable, aggregate-free integrity assertions, has been implemented and is described in detail in [SCHO75].

Briefly, integrity assertions are entered in the form of QUEL qualification clauses to be applied to interactions updating the relation over which the variable in the assertion ranges. A parse tree is created for the qualification and a representation of this tree stored in the INTEGRITY catalog together with an indication of the relation and specific domains involved. At query modification time, updates are checked for any possible integrity assertions on the affected domains. Relevant assertions are retrieved, re-built into tree form and grafted onto the update tree so as to AND the assertions with the existing qualification of the interaction.

4.3 PROTECTION AND VIEWS

Algorithms for the support of views are also given in [STON75]. Basically a view is any relation which could be created from existing relations by the use of a RETRIEVE command. Such view definitions will be treated in a manner somewhat analogous to that used for integrity control. They will be allowed in INGRES to support ECUEL programs written for obsolete versions of the data base and for user convenience.

Protection will be handled according to the algorithm described in [STON74b]. Like integrity control this algorithm involves

adding qualifications to the user's interaction. In the remainder of this section we distinguish this protection scheme from the one in [CHAM75] and indicate the rationale behind its use.

Consider the following two views:

```
RANGE OF E IS EMPLOYEE
DEFINE RESTRICTION-1 (E.NAME, E. SALARY, E.AGE)
WHERE E.DEPT = "toy"

DEFINE RESTRICTION-2 (E.NAME, E.DEPT, E.SALARY)
WHERE E.AGE < 50
```

and the following two access control statements:

```
RANGE OF E IS EMPLOYEE
PROTECT (E.NAME, E.SALARY, E.AGE)
WHERE E.DEPT = "toy"

PROTECT (E.NAME, E.SALARY, E.DEPT)
WHERE E.AGE < 50
```

Acess control could be based on views as suggested in [CHAM75] and a given user could be authorized to use views RESTRICTION-1 and RESTRICTION-2. To find the salary of Harding he could interrogate RESTRICTION-1 as follows:

```
RANGE OF R IS RESTRICTION-1
RETRIEVE (R.SALARY) WHERE
R.NAME = "Harding"
```

Failing to find Harding in RESTRICTION-1 he would have to then interrogate RESTRICTION-2. After two queries he would be returned the appropriate salary if Harding was under 50 or in the toy department.

Under the INGRES scheme the user can issue

```
RANGE OF E IS EMPLOYEE
RETRIEVE (E.SALARY) WHERE
E.NAME = "Harding"
```

which will be modified by the access control algorithm to

```
RANGE OF E IS EMPLOYEE
RETRIEVE (E.SALARY) WHERE
E.NAME = "Brown"
    AND
(E.AGE < 50 OR E.DEPT = "toy")
```

In this way the user need not manually sequence through his views to obtain desired data but automatically obtains such data if permitted. Note clearly that the portion of EMPLOYEE to which the user has access (the union of RESTRICTION-1 and RESTRICTION-2) is not a relation and hence cannot be defined as a single view.

To summarize, access control restrictions are handled automatically by the INGRES algorithms but must be dealt with by a user sequencing through his views in a "view oriented" access control scheme.

4.4 CONCURRENCY CONTROL

In any multiuser system provisions must be included to ensure that multiple concurrent updates are executed in a manner such that some level of data integrity can be guaranteed. The following two (somewhat facetious) updates illustrate the problem.

```
        RANGE OF E is EMPLOYEE
U1      REPLACE E(DEPT = "toy") WHERE
                E.DEPT = "candy"
```

```
        RANGE OF F is EMPLOYEE
U2      REPLACE F(DEPT = "candy") WHERE
                F.DEPT = "toy"
```

If U1 and U2 are executed concurrently with no controls, some employees may end up in each department and the particular result may not be repeatable if the data base is backed up and the interactions reexecuted.

The control which must be provided is to guarantee that some data base operation is "atomic" (i.e. occurs in such a fashion that it appears instantaneous and before or after any other data base operation). This atomic unit will be called a transaction.

In INGRES there are three basic choices available for defining a transaction.

a) something smaller than one INGRES command

b) one INGRES command

c) a collection of INGRES commands.

If a) is chosen INGRES could not guarantee that two concurrently executing update commands gave the same result as if they were executed sequentially (in either order) in one collection of INGRES processes. In fact the outcome could fail to be repeatable, as noted in the example above. This situation is clearly undesirable.

Option c) is in the opinion of the INGRES designers impossible to

support. The following transaction could be declared in an EQUEL program.

```
BEGIN TRANSACTION
     FIRST QUEL UPDATE
     SYSTEM CALLS TO CREATE AND DESTROY FILES
     SYSTEM CALLS TO FORK A SECOND COLLECTION OF INGRES
     PROCESSES TO WHICH COMMANDS ARE PASSED
     SYSTEM CALLS TO READ FROM A TERMINAL
     SYSTEM CALLS TO READ FROM A TAPE
     SECOND QUEL UPDATE (whose form depends on previous two
     system calls)
END TRANSACTION
```

Suppose T1 is the above transaction and runs concurrently with a transaction T2 involving commands of the same form. The second update of each transaction may well "conflict" with the first update of the other. Note that there is no way to tell apriori that T1 and T2 conflict because the form of the second update is not known in advance. Hence a deadlock situation can arise which can only be resolved by aborting one transaction (an undesirable policy in the eyes of the INGRES designers) or attempting to back out one transaction. The overhead of backing out through the intermediate system calls appears prohibitive (if it is possible at all). Restricting a transaction to have no system calls (and hence no I/O) cripples the power of a transaction in order to make deadlock resolution possible and was judged undesirable. Thus option b) was chosen.

The implementation of b) can be achieved by physical locks on data items, pages, tuples, domains, relations, etc. [GRAY75] or by predicate locks [STON74c]. The current implementation is by relatively crude physical locks (on domains of a relation) and

avoids deadlock by not allowing an interaction to proceed to process 3 until it can lock all required resources. Because of a problem with the current design of certain access method calls, all domains of a relation must currently be locked (i.e. a whole relation is locked) to perform an update. This situation will soon be rectified.

The choice of avoiding deadlock rather than detecting and resolving it is made primarily for implementation simplicity. The choice of a crude locking unit reflects a minicomputer environment where core storage for a large lock lable is not available. In the future we plan to experimentally implement a crude and (thereby low CPU overhead) version of a predicate locking scheme previously described in [STON74c]. Such an approach may provide considerable concurrency at an acceptable overhead in lock table space and CPU time, although such a statement is highly speculative.

Once the concurrency processor has assembled locks on all domains needed by an interaction, it may proceed to process 3 for unencumbered execution.

To conclude this section we briefly indicate the reasoning behind not sorting a page and its overflow pages in the "ISAM-like" access method. This topic is also discussed in [HELD75c].

Basically, maintenance of the sort order of these pages may require the access method to lock more than one page when it inserts a tuple. Clearly deadlock might be possible given

concurrent updates and locks for physical pages would be required (at least once a more sophisticated predicate locking scheme is tried such as [STON74c]). To avoid both problems these pages remain unsorted and the access method need only be able to read-modify-write a single page as an atomic operation. Although such an atomic operation is not currently in UNIX (and not needed by the current primitive scheme) it is a minor addition.

5 PROCESS 3

As noted in Section 2 this process performs the following two functions which will be discussed in turn:

a) the DECOMPOSITION of queries involving more than one variable into sequences of one-variable queries. Partial results are accumulated until the entire query is evaluated. This program is called DECOMP. It also turns any updates into the appropriate queries to isolate qualifying tuples and spools new values into a special file for deferred update.

b) the processing of a single variable queries. The program is called the one variable query processor (OVQP).

5.1 DECOMP

Because INGRES allows interactions which are defined on the cross product of perhaps several relations, efficient execution of this step is of crucial importance in order to search as small a portion of the appropriate cross product space as possible. DECOMP uses three techniques in processing interactions. We describe each technique then give the actual algorithm implemented. Finally, we indicate the role of a more sophisticated decomposition scheme under design.

a) Tuple substitution

The basic technique used by DECOMP to reduce a query to fewer

variables is tuple substitution. One variable (out of possibly many) in the query is selected for substitution. The AMI language is used to scan the relation associated with the variable one tuple at a time. For each tuple, the values of domains in that relation are substituted into the query. In the resulting modified query, all previous references to the substituted variable have now been replaced by values (constants), and the query has thus been reduced to one less variable. Precomposition is repeated (recursively) on the modified query until only one variable remains, at which point the OVQP is called to continue processing.

b) One-Variable Detachment

If the qualification Q of the query is of the form

Q1(V1) AND Q2(V1,...,Vn)

for some tuple variable V1, the following two steps can be executed:

1) Issue the query

```
RETRIEVE INTO W (TL[V1])
WHERE Q1[V1]
```

Here TL[V1] are those domains required in the remainder of the query. Note that this is a one variable query and may be passed directly to OVQP.

2) Replace R1, the relation over which V1 ranges, by W in the range declaration and delete Q1[V1] from Q.

The query formed in 1) is called a "one-variable, detachable sub-query" (OVDSQ) and the technique for forming and executing it "one-variable detachment" (OVD). This step has the effect of reducing the size of the relation over which V1 ranges by restriction and projection. Hence, it may reduce the complexity of the processing to follow.

Moreover, the opportunity exists in the process of creating new relations through OVD, to choose storage structures (and particularly keys) which will prove helpful in further processing.

c) Reformatting

When a tuple variable is selected for substitution, a large number of queries each with one less variable will be executed. If b) is a possible operation after the substitution for some remaining variable, V1, then the relation over which V1 ranges, R1, can be reformatted to have domains used in Q1(V1) as a key. This will expedite b) each time it is executed during tuple substitution.

We can now state the complete decomposition algorithm.

a) If number of variables in query is 0 or 1 call OVOP and stop; else go on.

b) Find all variables, {V1,...Vn}, for which the query contains a one-variable clause. Perform OVD to create new ranges for each of these variables. The new relation for each variable, Vi, is

stored as a hash file with key, Ki, chosen as follows.

1) For each j select from the remaining multi-variable clauses in the query the collection, C(ij), which have the form

Vi.di = Vj.dj

where di,dj are domains of Vi and Vj.

2) Form the key Ki to be the concatenation of domains di1,di2,...of Vi appearing in clauses in C(ij).

3) If more than one j exists, for which C(ij) is non empty, one C(ij) is chosen arbitrarily for forming the key. If C(ij) is empty for all j, the relation is stored as an unsorted table.

c) Choose the variable, Vs, with the smallest number of tuples as the next one for which to perform tuple substitution.

d) For each tuple variable Vj for which C(js) is non null, reformat the storage structure of the relation Rj which it ranges over, if necessary, so that the key of Rj is the concatenation of domains dj1,... appearing in C(js). This ensures that when the clauses in C(js) become one-variable after substituting for Vs, subsequent calls to OVQP to restrict further the range of Vj will be done as efficiently as possible.

e) Perform the following two steps for all tuples in the range of the variable selected in (c):

1) substitute values from tuple into query.

2) call decomposition algorithm recursively on a copy of resulting query which now has been reduced by one variable.

The following comments on the algorithm are appropriate:

a) OVD is almost always assured of speeding processing. Not only is it possible to wisely choose the storage structure of a temporary relation but also the cardinality of this relation may be much less than the one it replaces as the range for a tuple variable. It only fails if little or no reduction takes place and reformating is unproductive.

It should be noted that a temporary relation is created rather than a list of qualifying tuple-id's. The basic tradeoff is that OVD must copy qualifying tuples but can remove duplicates created during the projection. Storing tuple-id's avoids the copy operation at the expense of reaccessing qualifying tuples and retaining duplicates. It is clear that cases exist where each strategy is superior. The INGRES designers have chosen OVD because it does not appear to offer worse performance than the alternative, allows a more accurate choice of the variable with the smallest range in step c) above and results in cleaner code.

b) Tuple substitution is done when necessary on the variable associated with the smallest number of tuples. This has the effect of reducing the number of eventual calls on OVQP.

c) Reformatting is done (if necessary) with the knowledge that

it will replace a collection of complete sequential scans of a relation by a collection of limited scans. This will almost always reduce processing time.

d) It is believed that this algorithm efficiently handles a large class of interactions. Moreover, the algorithm does not require excessive CPU overhead to perform. There are, however, cases where a more elaborate algorithm is needed. The following comment applies to these cases.

e) Suppose that we have two or more strategies ST0, ST1, ... ,STn, each one being better than the previous one but also requiring a greater overhead. Suppose we begin an interaction on ST0 and run it for an amount of time equal to a fraction of the estimated overhead of ST1. At the end of that time, by simply counting the number of tuples of the first substitution variable which have already been processed, we can get an estimate for the total processing time using ST0. If this is significantly greater than the overhead of ST1, then we switch to ST1. Otherwise we stay and complete processing the interaction using ST0. Obviously, the procedure can be repeated on ST1 to call ST2 if necessary, and so forth.

The algorithm detailed in this section is ST0. A more sophisticated algorithm ST1 is currently under development and is discussed in [WONG76].

5.2 ONE VARIABLE QUERY PROCESSOR (OVQP)

This program is concerned solely with the efficient accessing of tuples from a single relation given a particular one-variable query. The initial portion of this program, known as STRATEGY, determines what key (if any) may be profitably used to access the relation, what the value(s) of that key will be used in calls to the AMI routine "find", and whether the access may be accomplished directly through the AMI to the storage structure of the relation itself or if a secondary index on the relation should be used. If access is to be through a secondary index then STRATEGY must choose which ONE of possibly many indices to use.

Then, the tuples retrieved according to the access strategy selected are processed by the SCAN portion of OVQP. This program evaluates each tuple against the qualification part of the query, creates target list values for qualifying tuples, and disposes of the target list appropriately.

Since SCAN is relatively straightforward, we discuss only the policy decisions made in STRATEGY.

First STRATEGY examines the qualification for clauses which specify the value of a domain, i.e. clauses of the form:

V.domain op constant

where "op" is one of {=, <, >, <=, >=}. Such clauses are termed "simple" clauses and are organized into a list

Obviously a non-simple clause may be equivalent to a simple one. For example

E.SALARY/2 = 10000 is equivalent to E.SALARY = 20000. However, recognizing and converting such clauses requires a general algebraic symbol manipulator. This issue has been avoided by ignoring all non-simple clauses. STRATEGY must now select one of two accessing strategies:

a) issuing two ANI find commands on the primary relation followed by a sequential scan of the relation between the limits specified

b) issuing two ANI find commands on some index relation followed by a sequential scan of the index between the limits specified. For each tuple retrieved the "pointer" domain is obtained and is a tuple-id of a tuple in the primary relation. This tuple is fetched and examined.

Key information about the primary relation is obtained using the ANI function "paramd". Names of indices are obtained from the index catalog and keying information about indices is obtained with the function "parami".

STRATEGY now checks if a simple clause is available to limit the scan of the primary relation or an index relation. If a relation is hashed the simple clause must specify equality as the operator in order to be useful. ISAM structures on the other hand allow ranges of values and less than all keys may be specified as long as the first one is present for structures with combined keys.

STRATEGY checks for such a simple clause for a relation in the following order:

a. hashed primary relation

b. hashed index

c. ISAM primary relation

d. ISAM index

The rationale for this ordering is related to the expected number of page accesses required to retrieve a tuple from the source relation in each case.

In case a) the key value provided locates a desired source tuple in one access, (ignoring overflows). In case b) the key value locates an appropriate index relation tuple in one access but an additional access in required to retrieve the proper source tuple. For the ISAM scheme, the directory must be examined. The number of accesses incurred in this look-up is at least 2. Again, with an index, an additional access is required, making the total at least 3 in case d).

6 UTILITIES IN PROCESS 4

6.1 IMPLEMENTATION OF UTILITY COMMANDS

We have indicated in Section 1 several data base utilities available to users. We will now briefly describe their implementation and indicate their configuration within the INGRES system.

The commands are organized into several overlay programs. Since an overlay may have more than one entry point, it can contain more than one utility command. In fact, the utilities are grouped where possible to minimize overlaying. The overlays all contain a common main program known as the "controller", which reads pipe C and writes completion messages into pipe D. The processing of a utility command occurs as follows.

First, the parser recognizes a utility command in a user interaction. This name is looked up in the INGRES "process-table", which has an entry for each command name in the language. Each entry has an "overlay-id" and a "function-id". The first indicates the overlay program containing the command, and, the second indicates the proper entry point within that overlay.

These id's are passed down pipe B to process 3 followed by the parameters of the command specified. Process 3 determines from the "overlay-id" if the command is a utility command intended for process 4. If so, the information is simply written on pipe C.

At this point, some overlay is occupying process 4, having remained from a previous command. Its copy of the controller

reads pipe C to obtain the overlay-id. If a different overlay from the present one is indicated, the controller overlays process 4 and control passes to the controller of the new overlay.

Most of the utilities update or read the system relations using AMI calls. MODIFY contains a sort routine which puts tuples in collating sequence according to the concatenation of the desired keys (which need not be of the same data type). Pages are initially loaded to approximately 80% of capacity. The sort routine is a recursive N-way merge-sort where N is maximum number of files process 4 can have open at once (currently 8). The index building occurs in an obvious way. To convert to hash structures MODIFY must specify the number of primary pages to be allocated. This parameter is used by the AMI in its hash scheme (which is a standard modulo division method). This is done by a rule of thumb.

It should be noted that a user who creates an empty hash relation using the CREATE command and then copies a large UNIX file into it using COPY will create a very inefficient structure. This is because a relatively small default number of primary pages will have been specified by CREATE and overflow chains will be long. A better strategy is to COPY into an unsorted table so that MODIFY can subsequently make a good guess at the number of primary pages to allocate.

6.2 DEFERRED UPDATE AND RECOVERY

Any updates (APPEND, DELETE, REPLACE) are processed by writing

the tuples to be added changed or modified into a temporary file. When process 3 finishes, it calls process 4 to actually perform the modifications requested as a final step in processing. Deferred update is done for four reasons.

a) Secondary index considerations. Suppose the following QUEL statement is executed;

```
RANGE OF E IS EMPLOYEE
REPLACE E(SALARY = 1.1*E.SALARY)  WHERE
     E.SALARY > 20000
```

Suppose further that there is a secondary index on the salary domain and the primary relation is keyed on another domain.

OVQP in finding the employees who qualify for the raise will use the secondary index. If one employee (say Smith qualifies and his tuple is modified and the secondary index updated) then the scan of the secondary index will find his tuple a second time (in fact an arbitrary number of times). Either secondary indexes cannot be used to identify qualifying tuples when range qualifications are present (a rather unnatural restriction) or secondary indices must be updated in deferred mode.

b) Primary relation considerations. Suppose the following QUEL statement is executed

```
RNGE OF E, M IS EMPLOYEE
REPLACE E(SALARY = .9*E.SALARY)  WHERE
     E. MGR = M.NAME
       AND
     E.SALARY > M.SALARY
```

for the following EMPLOYEE relation

NAME	SAL	MANAGER
Smith	10k	Jones
Jones	8k	
Brown	9.5k	Smith

Logically Smith should get the pay cut but Brown should not. However, if Smith's tuple is updated before Brown is checked for the pay cut, Brown will qualify. This undesirable situation must be avoided by deferred update.

c) Functionality of updates. Suppose the following QUEL statement is executed.

```
RANGE of E,M is EMPLOYEE
REPLACE E(SALARY = M.SALARY)
```

This update attempts to assign to each employee the salary of every other employee, i.e. a single data item is to be replaced by multiple values. Stated differently, the REPLACE statement does not specify a function. This non-functionality can only be checked if deferred update is performed.

d) Recovery is easier. The deferred update file provides a log of updates to be made. Recovery is provided upon system crash by the RESTORE command. In this case the deferred update routine is requested to destroy the temporary file if it has not yet started processing it. If it has begun processing, it reprocesses the entire update file which is done in such a way that the effect is the same as if it were processed exactly once from start to finish.

Hence the update is "backed out" if deferred updating has not yet begun; otherwise it is processed to conclusion. The software is

designed so the update file can be optionally spooled onto tape and recovered from tape. This added feature should soon be operational.

If a user from the terminal monitor (or a C-program) wishes to stop a command he can issue a "break" character. In this case all processes reset execept the deferred update program which recovers in the same manner as above.

All update commands do deferred update; however the INGRES utilities have not yet been modified to do likewise. When this is completed INGRES will recover from all crashes which leave the disk intact. In the meantime there can be disk-intact crashes which cannot be recovered in this manner (if they happen in such a way that the system catalogs are left inconsistent).

The INGRES "super-user" can checkpoint a data base(s) onto tape using the UNIX backup scheme. Since INGRES logs all interactions, a consistent system can always be obtained (albeit slowly) by restoring the last checkpoint and running the log of interactions (or the tape(s) of deferred updates if it exists).

7 CONCLUSION AND FUTURE EXTENSIONS

The system described herein is in use at 5 installations and is being brought up at 8 others. It forms the basis of an accounting system, a system for managing student records, a geo-data system, a system for maintaining a wiring diagram for a large telelphone company and assorted other smaller applications. These applications have been running for periods of up to nine months.

7.1 PERFORMANCE

At this time no detailed performance measurements have been made. However, on our system (an 11/40 mainframe with 80k words of core) about 4-6 simultaneous INGRES users can be supported with reasonable response time (assuming they are doing interactions which affect a small number of tuples and for which a fast access path exists. Of course, a user has the ability to execute interactions in INGRES which require hours to process). This hardware configuration costs about $60,000. Larger 11/70 installations should be able to run substantially more INGRES users.

The sizes of the processes in INGRES are indicated below. Since the access methods are loaded with processes 2 and 3 and with many of the utilities their contribution to the respective process sizes has been noted separately.

access methods (AM)	11 K (bytes)
terminal monitor	10 K

EQUEL	30 K + AM
process 2	45 K + AM
process 3 (query processor)	45 K + AM
utilities (8 overlays)	160 K + AM

7.2 USER FEEDBACK

The feedback from internal and external users has been overwhelmingly positive. In this section we indicate features that have been suggested for future systems.

a) higher performance

Earlier versions of INGRES were very slow. The current version should alleviate this problem.

b) recursion

QUEL does not support recursion. Hence, recursion must be tediously programmed in C using the precompiler. This has been suggested as a desired extension.

c) other language extensions

These include user defined functions (especially counters), multiple target lists for a single qualification statement and if-then-else control structures in QUEL. These extensions are also a user can avoid using the precompiler.

d) report generator

PRINT is only a very primitive report generator. The need for augmented facilities in this area is clear. It should be written in EQUEL.

e) bulk copy

The COPY routine fails to handle easily all situations that have arisen.

7.3 FUTURE EXTENSIONS

Noted throughout the paper are areas where system improvement is in progress, planned or desired by users. Other areas of extension include the following

a) A multi-computer system version of INGRES to operate on distributed data bases

b) Further performance enhancements

c) A higher level user language including recursion and user defined functions

d) Better data definition and integrity features

e) A data base administrator advisor. This program would run at idle priority and issue queries against a statistics relation to be kept by INGRES. It could then offer advice to a DBA concerning the choice of access methods and the selection of indices. This topic is discussed further in [HELD75b].

ACKNOWLEDGEMENT

REFERENCES

ALLM76 Allman, E., Stonebraker, M., and Held, G. "Embedding a Relational Data Sub-language in a General Purpose Programming Language.", Proc. ACM SIGPLAN-SIGMOD Workshop on Data, Salt Lake City, Utah, March, 1976.

BOYC73 Boyce, R. et. al., "Specifying Queries as Relational Expressions: SQUARE", IBM Research, San Jose, Ca., RJ 1291, Oct. 1973.

CHAM74 Chamberlin, D. and Boyce, R., "SEQUEL: A Structured English Query Language", Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

CHAM75 Chamberlin, D., Gray, J.N., and Traiger, I.L., "Views, Authorization and Locking in a Relational Data Base System", Proc. 1975 National Computer Conference, AFIPS Press, May 1975.

CODA71 Committee on Data Systems Languages, "CODASYL Data Base Task Group Report", ACM, New York, 1971.

CODD70 Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM, Vol. 13 No. 6, pp. 377-387, June, 1970.

CODD71 Codd, E.F., "A Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control, San Diego, Ca., Nov. 1971.

CODD72 Codd, E.F., "Relational Completeness of Data Base Sublanguages", Courant Computer Science Symposium 6, May 1972.

CODD74 Codd, E.F. and Date, C.J., "Interactive Support for Non-Programmers, The Relational and Network Approaches",

Proc. 1974 ACM-SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

DATE74 Date, C.J. and Codd, E.F., "The Relational and Network Approaches: Comparison of the Aplication Programming Interfaces", Proc. 1974 ACM-SIGFIDET Worksnop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

GRAY75 Gray, J.N., Lorie, R.A., and Putzolu, G.R. "Granularity of Locks in a Shared Data Base" Proc. International Conference on Very Large Data Bases, Framingham, Mass., September, 1975.

GO75 Go, A., Stonebraker, M., and Williams, C., "An Approach to Implementing a Geo-data System", Proc. ACM SIGGRAPH/SIGMOD Conference on Data Bases in Interactive Design, Waterloo, Ontario, Sept. 1975.

GOTT75 Gottlieb, D., et. al. "A Classification of Compression Methods and their Usefulness in a Large Data Processing Center" Proc. 1975 National Computer Conference, AFIPS Press, May, 1975.

HELD75a Held, G.D., Stonebraker, M. and Wong, E., "INGRES - A Relational Data Base Management System", Proc. 1975 National Computer Conference, AFIPS Press, 1975.

HELD75b Held, G.D., "Storage Structures for Relational Data Base Management Systems", Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, 1975.

HELD75c Held, G., and Stonebraker M., "B-Trees Re-examined", to appear in CACM.

IBM66 IBM Corp., "OS ISAM Logic", IBM, White Plains, N.Y., GY28-6618.

JOHN74 Johnson, S.C., "YACC, Yet Another Compiler-Compiler", UNIX Programmer's Manual, Bell Telephone Labs, Murray Hill, N.J., July 1974.

MCDO75a McDonald, N. and Stonebraker, M., "Cupid -- The Friendly Query Language", Proc. ACM-Pacific-75, San Francisco, California, April 1975.

MCDO75b McDonald, Nancy., "CUPID: A Graphics Oriented Facility for Support of Non-programmer Interactions with a Data Base", Ph.D. Thesis, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1975.

RITC74 Ritchie, D.M. and Thompson, K. "The UNIX Time-Sharing System," CACM, Vol. 17, No. 3., March, 1974.

SCHO75 Schoenberg, Iris, "Implementation of Integrity Constraints in the Relational Data Base Management System, INGRES", M.S. Thesis, Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1975.

STON74a Stonebraker, M., "A Functional View of Data Independence" Proc. 1974 SIGFIDET Workshop on Data Description, Access and Control, Ann Arbor, Mich., May 1974.

STON74b Stonebraker, M. and Wong, E., "Access Control in a Relational Data Base Management System by Query Modification", Proc. 1974 ACM National Conference, San Diego, Ca., Nov. 1974

STON74c Stonebraker, M., "High Level Integrity Assurance in Relational Data Base Systems", University of California, Electronics Research Laboratory, Memo ERL-M473, August, 1974.

STON75 Stonebraker, M., "Implementation of Integrity Constraints

and Views by Query Modification", Proc 1975 SIGMOD Workshop on Management of Data, San Jose, Ca., May 1975.

TSIC75 Tsichritzis, D., "A Network Framework for Relational Implementation", University of Toronto, Computer Systems Research Group Report CSRG-51, Feb. 1975

WONG76 Wong, E. and Youssefi, K., "Decomposition-A Strategy for Query Processing" (Submitted)

ZOOK75 Zook, W., Youssefi, K., Kreps, P., Held, G. and Ford, J., "INGRES - Reference Manual", University of California, Electronics Research Laboratory, Memo. No. ERL-M519, April, 1975.